Kubernetes Fundamentals

A sample chapter from LFS258 Kubernetes Fundamentals, the online self-paced Linux Foundation course.

March 2017
A publication of The Linux Foundation Training
training.linuxfoundation.org

THE LINUX FOUNDATION

TRAINING

Basics of Kubernetes and Kubernetes Architecture are the first two sections in The Linux Foundation's online, self-paced course, Kubernetes Fundamentals (LFS258).

This ebook, which contains both sections, gives a high-level overview of what Kubernetes is and the challenges it solves, and then dives deeper into the system architecture. Subsequent chapters cover the API and its most important resources, how to deploy and manage an application on Kubernetes, and some upcoming features that will boost your productivity.

In its entirety, the course provides Linux administrators and software developers who are starting to work with containers the key principles behind managing containerized applications in production.

Here you will find:

- A course overview
- An introduction to the course instructor
- 44 pages of concepts from two chapters of the course
- A lab exercise on Docker networking and how to start a Kubernetes head node

The online course content includes video recordings. Therefore, this ebook includes the images and text that are on each page of the course chapters, as well as the audio transcript for the videos or links to the screencasts on YouTube.

This sample chapter is intended to give an overview of the course format and quality of the content. The course is prepared and presented by Sebastien Goasguen, founder of Skippbox, a Kubernetes startup that provides solutions, services, and training for this key cloud-native technology.

Table of Contents

1 Course Introduction

Course Introduction

This page has a video introducing the course. A transcript of the video is below. Note that the forums are only available to students enrolled in the course.

This course is about Kubernetes, which is an open source system to manage containerized applications. Kubernetes was created by Google, and since it's been open sourced by Google there's been an enormous community that has grown around Kubernetes. So it's becoming almost the de facto system to manage containerized applications.

That's why this course is so important. If you embrace containers and start running containerized applications in your data center you'll need a system like Kubernetes to manage them, scale them, update them, and so on.

This course is for developers and system administrators. You'll learn the basics of Kubernetes. I really focus on the API and all the primitives needed to build distributed applications. So you'll learn about pods and services and deployments that are all API-driven and defined in manifests. You'll learn how to create those primitives, those objects, and how they make up your entire distributed application.

I'll see you on the forums where you'll be able to ask all your questions. I hope you enjoy the course and have a great learning experience. Let's get started!

Course Learning Objectives

By the end of this course, you will learn the following:

- **The history and evolution of Kubernetes.**
- **Its high-level architecture and components.**
- **The API and the most important resources that make the API.**
- **How to deploy and manage an application.**
- **Some upcoming features that will boost your productivity.**

Course Audience and Requirements

If you are a Linux administrator or software developer starting to work with containers and wondering how to manage them in production, **LFS258: Kubernetes Fundamentals** is the course for you.

In this course, you will learn the key principles that will put you on the journey to managing containerized applications in production.

To make the most of this course, you will need the following:

- A good understanding of Linux
- Familiarity with the command line
- Familiarity with package managers
- Familiarity with Git and GitHub
- Access to a Linux server or Linux desktop/laptop
- VirtualBox on your machine, or access to a public cloud.

Meet Your Instructor: Sebastien Goasguen

Sebastien Goasguen is a 15 years open source veteran. After a career in academia that led him from Arizona, to Indiana and South Carolina, he joined the computer industry and worked on Infrastructure as a Service solutions. He is a regular speaker at open source conferences, and an active blogger on linux.com. In 2014, Sebastien joined the Docker movement, impressed by its user experience and ease of building new applications. He wrote the O'Reilly *Docker Cookbook* and, while investigating the Docker ecosystem, he started focusing on Kubernetes. He is now the founder of Skippbox, a Kubernetes startup that provides solutions, services, and training for this key Cloud-native technology. He is a member of the Apache Software Foundation, and a member/contributor to the Kubernetes Incubator.

2 Basics of Kubernetes

CHAPTER 2:
BASICS OF KUBERNETES

Audio Transcript:

In this chapter, *Basics of Kubernetes*, we are going to spend some time looking at what Kubernetes is, at a very high level, we are going to talk about the challenges that it solves, and introduce a few other solutions that you may consider. We are going to talk about Borg, which is the internal solution used at Google to manage distributed applications, and we are going to also look at the Kubernetes lineage, and talk about some container technologies that have been developed by Google. We'll look at the Kubernetes architecture, and basic components that run on the master node and on the worker nodes, and we'll also look at how Kubernetes is doing as an open source project. We'll look at a few Kubernetes users and case studies that you can find on the Kubernetes website, and then, we will look at the Cloud Native Computing Foundation, which is the foundation that governs the Kubernetes project.

Learning Objectives

By the end of this chapter, you should be able to:

- **Define Kubernetes.**
- **Explain where does Kubernetes come from.**
- **Discuss the adoption status of Kubernetes.**
- **Discuss the governance of Kubernetes as an open-source project.**

What is Kubernetes?

Kubernetes is "an open-source software for automating deployment, scaling, and management of containerized applications", according to the kubernetes.io website.

Running a container on a laptop is relatively simple. But connecting containers across multiple-hosts, scaling them when needed, deploying applications without downtime, and service discovery among several aspects, are really hard challenges.

Kubernetes addresses those challenges from the start with a set of primitives and a powerful API.

A key aspect of **Kubernetes** is that it builds on 15 years of experience at Google.

Google infrastructure started reaching high scale before virtual machines became pervasive in the datacenter and containers provided a fine grain solution to pack clusters efficiently. Efficiency in using clusters and managing distributed applications has been at the core of Google challenges.

<u>The Meaning of Kubernetes</u>

Kubernetes, in Greek **κυβερνήτης**, means the Helmsman, or pilot of the ship.

Keeping with the maritime theme of **Docker** containers, **Kubernetes** is the pilot of a ship of containers.

Challenges

Containers have seen a huge rejuvenation in the past three years. They provide a great way to package, ship, and run applications - that is the **Docker** motto. The developer experience has been boosted tremendously thanks to containers. Containers, and **Docker** specifically, have empowered developers with:

- Ease of building container images
- Simplicity of sharing images via **Docker** registries
- Powerful user experience to manage containers.

However, managing containers at scale and architecting a distributed application based on microservices' principles is still challenging.

You first need a continuous integration pipeline to build your container images, test them, and verify them. Then, you need a cluster of machines acting as your base infrastructure on which to run your containers. You also need a system to launch your containers, watch over them when things fail and self-heal. You must be able to perform rolling updates and rollbacks, and you need a network setup which permits self-discovery of services in a very ephemeral environment.

Other Solutions

Kubernetes, which we will be covering in this course, is definitely a solution to manage containerized applications. But there are other solutions as well:

- Docker Swarm is the Docker Inc. solution. It has been re-architected recently and is based on SwarmKit. It is embedded with the **Docker Engine**.
- Apache Mesos is a datacenter scheduler, which can run containers through the use of *frameworks*. Marathon is the framework that lets you orchestrate containers.
- Nomad from HashiCorp, the makers of **Vagrant** and **Consul**, is another solution for managing containerized applications. **Nomad** schedules tasks defined in *Jobs*. It has a **Docker** driver which lets you define a running container as a task.
- Rancher is a container orchestrator agnostic system, which provides a single pane of glass interface to managing applications. It supports **Mesos**, **Swarm**, **Kubernetes**, as well as its native system, **Cattle**.

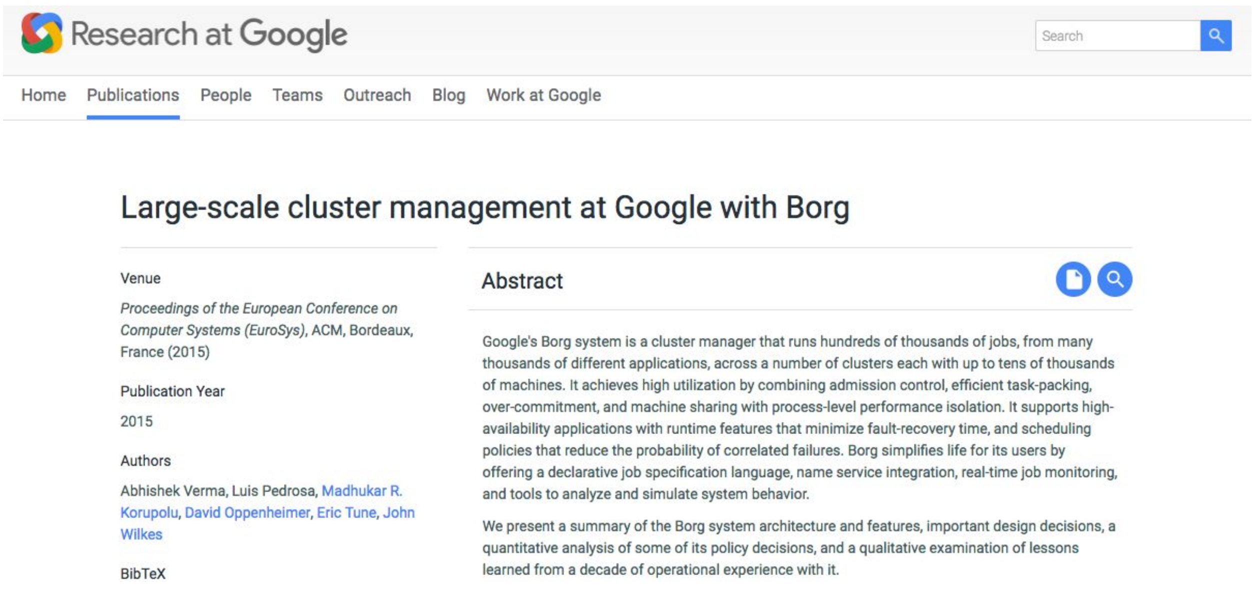
The Borg Heritage

What primarily distinguishes **Kubernetes** from other systems is its heritage. **Kubernetes** is inspired by **Borg** - the internal system used by Google to manage its applications (e.g Gmail, Apps, GCE).

With Google pouring the valuable lessons they learned from writing and operating **Borg** for over 15 years into **Kubernetes**, this makes **Kubernetes** a safe choice when having to decide on what system to use to manage containers.

Think about this: when you use Gmail, Google Docs, GCE or any other Google service, you are using a containerized application managed by Borg worldwide:

- **Borg** was a Google secret for a long time, but not anymore.
- **Borg** is the orchestration system to manage all Google applications at scale
- **Borg** was finally described publicly in 2015
- To learn more about the ideas behind **Kubernetes**, you can read the "Large-scale cluster management at Google with Borg" paper.



Large-scale cluster management at Google with Borg

Venue

Proceedings of the European Conference on Computer Systems (EuroSys), ACM, Bordeaux, France (2015)

Publication Year

2015

Authors

Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, John Wilkes

BibTeX

Abstract

Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines. It achieves high utilization by combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation. It supports high-availability applications with runtime features that minimize fault-recovery time, and scheduling policies that reduce the probability of correlated failures. Borg simplifies life for its users by offering a declarative job specification language, name service integration, real-time job monitoring, and tools to analyze and simulate system behavior.

We present a summary of the Borg system architecture and features, important design decisions, a quantitative analysis of some of its policy decisions, and a qualitative examination of lessons learned from a decade of operational experience with it.

Figure 2.1: The "Large-scale cluster management at Google with Borg" Paper (retrieved from http://research.google.com/pubs/pub43438.html)

Kubernetes Lineage

When talking about **Borg**, we also have to put things in perspective. This is best reflected by Figure 2.2, from the Cloud Foundry Foundation. Figure 2.2 shows how **Borg** has inspired current datacenter systems, as well as the underlying technologies used in container runtime today.

- Google contributed **`cgroups`** to the **Linux** kernel in 2007; it limits the resources used by collection of processes.
- **`cgroups`** and **Linux `namespaces`** are at the heart of containers today, including **Docker**.
- **Mesos** was inspired by discussions with Google when **Borg** was still a secret. Indeed, **Mesos** builds a multi-level scheduler, which aims to better use a datacenter cluster.
- Cloud Foundry Foundation embraces the 12 factor application principles. These principles provide great guidance to build web applications that can scale easily, can be deployed in the Cloud, and whose build is automated. **Borg** and **Kubernetes** address these principles as well.

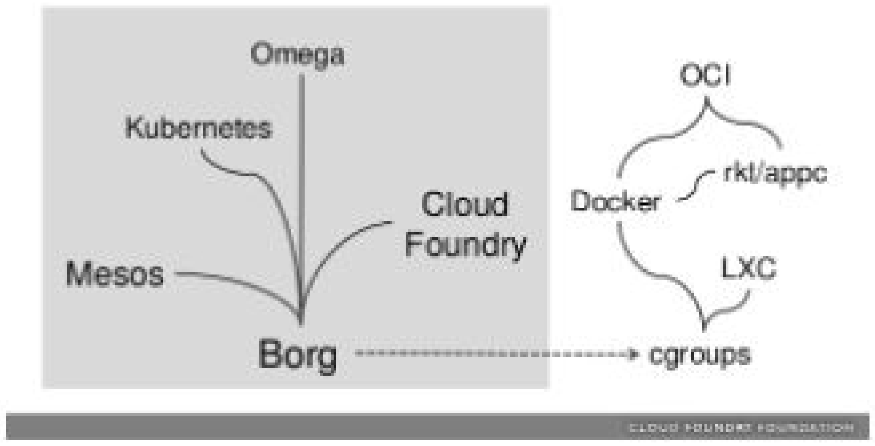


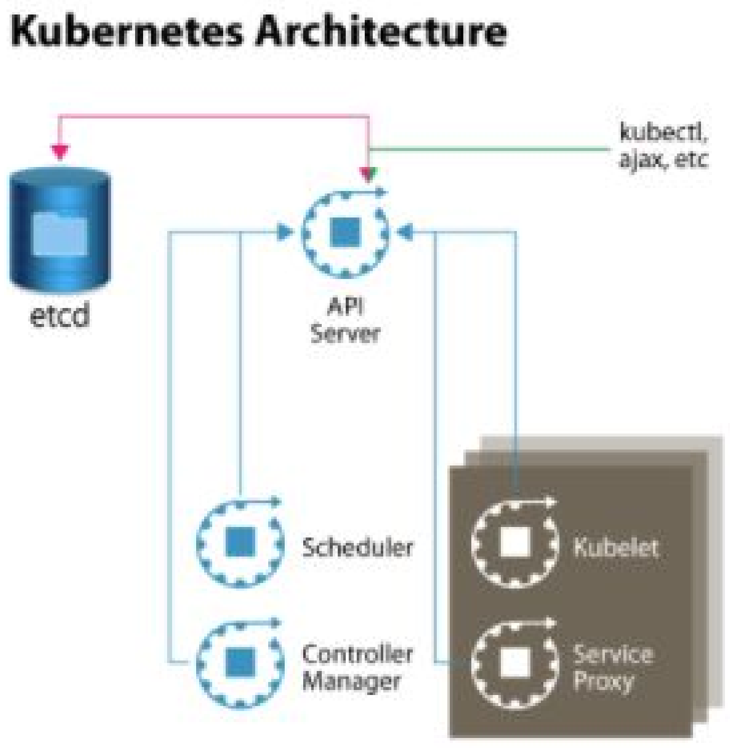
Figure 2.2: The Kubernetes Lineage (by **Chip Childers**, **Cloud Foundry Foundation**, retrieved from LinkedIn Slideshare)

<u>Kubernetes Architecture</u>

To quickly demistify **Kubernetes**, let's have a look at Figure 2.3, which shows a high-level architecture diagram of the system components.

In its simplest form, **Kubernetes** is made of a central manager (aka master) and some worker nodes (We will see in a follow-on chapter how you can actually run everything on a single node for testing purposes). The manager runs an API server, a scheduler, a controller and also a storage system to keep the state of the cluster.

Kubernetes exposes an API (via the API server): you can communicate with the API using a local client called *kubectl* or you can write your own client (green arrow). The scheduler sees the requests for running containers coming to the API and finds a suitable node to run that container in. Each node in the cluster runs two processes: a *kubelet* and a *service proxy*. The kubelet receives requests to run the containers and watches over them on the local node. The proxy creates and manages networking *rules* to expose the container on the network.



Kubernetes Architecture (Cont.)

In a nutshell, **Kubernetes** has the following characteristics:

- It is made of a manager and a set of nodes
- It has a scheduler to place containers in a cluster
- *It has an API server and a persistence layer with etcd*
- It has a controller to reconcile states
- It is deployed on VMs or bare-metal machines, in public Clouds, or on-premise
- It is written in Go.

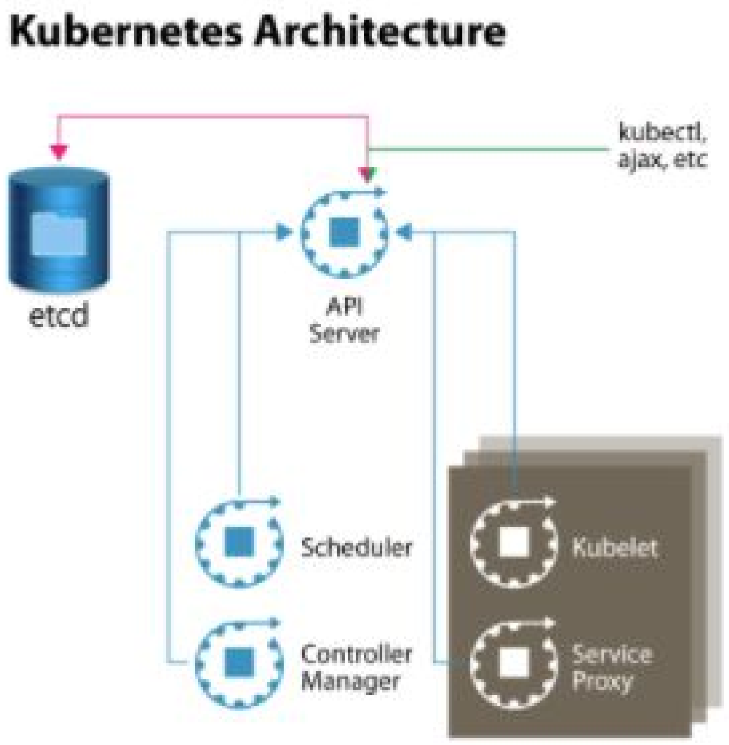


Figure 2.3: Kubernetes Architecture

How Is Kubernetes Doing?

Since its inception, **Kubernetes** has seen a terrific pace of innovation and adoption. The community of developers, users, testers, and advocates is continuously growing every day. The software is also moving at an extremely fast pace, which is even putting **GitHub** to the test.

Here are a few numbers:

- It was open sourced in June 2014
- It has 1000+ contributors
- There are 37k+ commits
- There have been meetups in over 100 cities worldwide, with over 30,000 participants in 25 countries
- There are over 8000 people on Slack
- There is one major release approximately every 3 months.

To see more interesting numbers about the **Kubernetes** community, you can check the infographic created by Apprenda.

Kubernetes Users

Kubernetes is being adopted at a very rapid page. To learn more, you should check out the case studies presented on the **Kubernetes** website. Ebay, box, Pearson and Wikimedia have all shared their stories.

Pokemon Go, the fastest growing mobile game, also runs on **Google Container Engine** (**GKE**), the **Kubernetes** service from **Google Cloud Platform** (**GCP**).

Figure 2.4 provides a snapshot of some of the **Kubernetes** users.

Figure 2.4: Kubernetes Users (by **Kubernetes**, retrieved from kubernetes.io)

The Cloud Native Computing Foundation

Kubernetes is an open source software with an Apache license. Google donated **Kubernetes** to a newly formed foundation within **The Linux Foundation** in July 2015, when **Kubernetes** reached the v1.0 release. This foundation is called the **Cloud Native Computing Foundation** (**CNCF**).

CNCF is not just about **Kubernetes**, it is serving as the governing body for open source software that solves specific issues faced by Cloud native applications (i.e applications that are written specifically for a Cloud environment).

CNCF has many corporate members that collaborate, such as Cisco, the Cloud Foundry foundation, AT&T, Box, Goldman Sachs, and many others.

Note: Since CNCF now owns the copyright of **Kubernetes**, contributors to the source need to sign a contributor license agreement (CLA) with CNCF, just like any contributor to an Apache-licensed project signs a CLA with the Apache Software Foundation.

CLOUD NATIVE COMPUTING FOUNDATION

Figure 2.5: Cloud Native Computing Foundation

A Few Bits To Remember

Before we move on to looking deeper into what **Kubernetes** (**k8s**) is, here are a few things to remember about it:

- It was inspired by Google's Borg
- It is open source and available on GitHub
- It is written in the Go programming language
- Its high-level architecture is made of a central manager running an API server and scheduler, and a set of nodes running an agent
- It is licensed under the Apache Software License v2.0
- It is governed by the Cloud Native Computing Foundation at The Linux Foundation
- There are several Special Interest Groups (SIG) that are open to everyone
- Weekly Hangouts take place. You can start with the community meeting every Thursday at 10 am PST.

Kubernetes Resource Recommendations

• Visit the Kubernetes website

• Find the Kubernetes source code on GitHub

• Read the famous Borg paper

• Read the more recent Borg, Omega, and Kubernetes paper

• Listen to John Wilkes talking about Borg and Kubernetes

• Add the Kubernetes community hangout to your calendar, and attend at least once

• Join the community on Slack and go in the #kubernetes-users channel.

• Check out the very active Stack Overflow community.

• Visit the Cloud Native Computing Foundation website and keep an eye on the conferences and various software.

Note: This course also includes a screencast that shows how to find these resources. You can access the video on YouTube.

Choose the correct answer.

On which of the following is Kubernetes based on?

a) Mesos

b) Borg

c) Maui

Choose the correct answer.

What language is Kubernetes written in?

a) Go

b) Python

c) C++

Choose the correct answer.

Where is the cluster state stored?

a) zookeper

b) MySQL

c) etcd

Learning Objectives (Review)

You should now be able to:

- **Define Kubernetes.**
- **Explain where does Kubernetes come from.**
- **Discuss the adoption status of Kubernetes.**
- **Discuss the governance of Kubernetes as an open-source project.**

3 Kubernetes Architecture

CHAPTER 3: KUBERNETES ARCHITECTURE

Audio Transcript:

In this chapter, *Kubernetes Architecture*, we are going to start diving a little deeper into Kubernetes, and check what the architecture of the system is. We are going to look at the services that run on the master node, we are going to look at the services that run on the worker nodes, and we'll look at how the state of the cluster is persisted, using **etcd**. We'll talk about the networking setup in Kubernetes, and we'll finish by looking at similar solutions, like Apache Mesos, and we'll highlight how they compare.

Learning Objectives

By the end of this chapter, you should be able to:

- **Discuss all the components that make up a Kubernetes cluster.**
- **Explain the similarity of a Kubernetes cluster to other cluster managers/orchestrators.**

High-Level Architecture

As mentioned in the previous chapter, a **Kubernetes** cluster is made of a master node and a set of worker nodes.

However, for testing purposes, all the components running in those nodes can be run on the same node, whether it is a virtual machine or a physical node. We will see more about this in the installation section when we use minikube.

Kubernetes has six main components that need to be running to get a functioning cluster:

- The API server
- The scheduler
- The controller manager
- The kubelet
- The proxy (aka *kube proxy*)
- An *etcd* cluster.

Each of these components can run as standard **Linux** processes on your nodes, or they can run as **Docker** containers themselves.

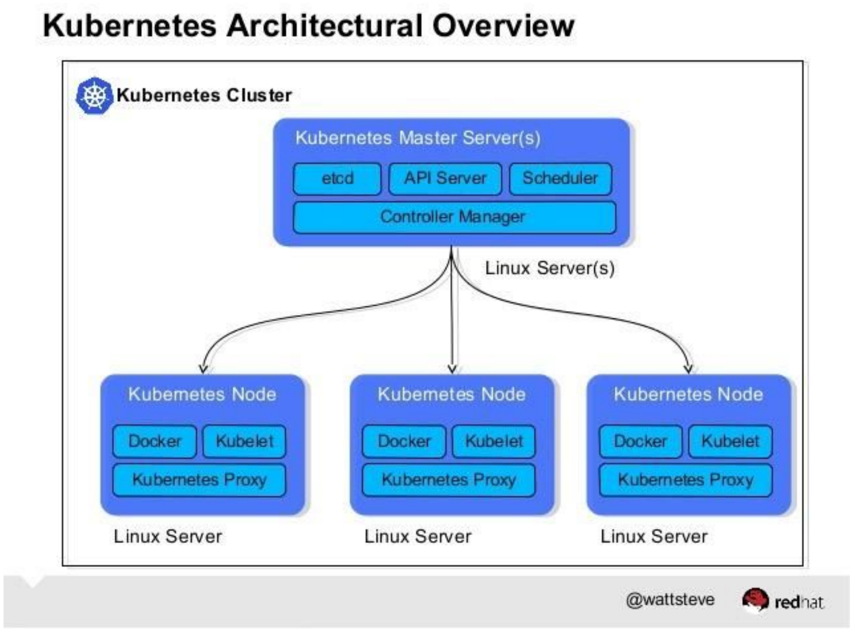


Figure 3.1: Kubernetes Architectural Overview (by **Steve Watt**, **RedHat**, retrieved from LinkedIn SlideShare)

The Master Node

The master node runs the API server, the scheduler, and the controller manager.

For example, on one of the **Kubernetes** master nodes that we started on a CoreOS instance, we see the following systemd unit files:

```
core@master ~ $ systemctl -a | grep kube

kube-apiserver.service            loaded active   running Kubernetes API Server
kube-controller-manager.service   loaded active   running Kubernetes Controller Manager
kube-scheduler.service            loaded active   running Kubernetes Scheduler
```

The API server exposes a REST interface to all of the **Kubernetes** resources and it is highly configurable.

The Scheduler's main responsibility is to place the containers on the node in the cluster according to various policies, metrics, and resource requirements. It is also configurable via command line flags.

Finally, the main controller, also called Controller Manager, is responsible for reconciling the actual state of the cluster with the desired state, as specified via the API. In effect, it is a control loop that performs actions based on the observed state of the cluster and the desired state. The main controller is also configurable.

The master node can be configured in a multi-master highly available setup. Indeed, the schedulers and controller managers can elect a leader, while the API servers can be fronted by a load-balancer.

The Worker Nodes

All the worker nodes run the *kubelet* and *kube proxy*, as well as the **Docker** engine.

The *kubelet* interacts with the underlying **Docker** engine also installed on all the nodes and makes sure that the containers that need to run are actually running. The `kube-proxy` is in charge of managing the network connectivity to the containers.

```
core@node-1 ~ $ systemctl -a | grep kube
kube-kubelet.service  loaded active  running Kubernetes Kubelet
kube-proxy.service    loaded active  running Kubernetes Proxy

core@node-1 ~ $ systemctl -a | grep docker
docker.service loaded active  running Docker Application Container Engine
docker.socket loaded active   running Docker Socket for the API
```

NOTE: You can also run an alternative to the Docker engine - *rkt* by **CoreOS**. To learn how you can do that, you should check the documentation. In future releases, it is highly likely that **Kubernetes** will support additional container runtime.

Keeping State with etcd

To know the state of the cluster over time, **Kubernetes** needs a persistency layer. Traditionally, this could be implemented with a relational database. However, in a highly scalable system, a relational database (e.g. **MySQL**, **PostgreSQL**) may become a single point of failure.

Distributed key-value stores are, by design, made to run on multiple nodes. Data is replicated among the nodes and has strong consistency. They are fault-tolerant in the sense that, upon machine failure, they will still continue to work. **Zookeeper**, **Consul**, and **etcd** are all examples of distributed key-value stores.

Kubernetes uses etcd. It can be run on a single node (even though it loses its distributed characteristic and hence its tolerance to failure). Underneath, *etcd* uses a leader election algorithm to provide strong consistency of the stored state among the nodes. A discussion about *etcd* is outside of the scope of this course, but you should however become familiar with its setup and operation.

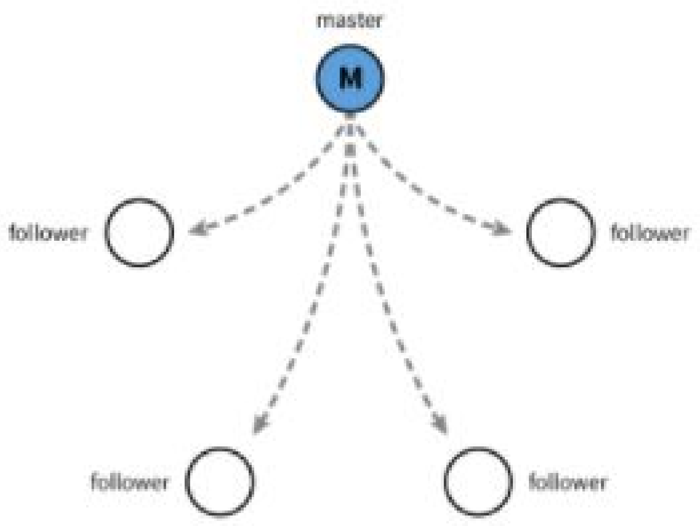


Figure 3.2: Logs replicated to each follower in the cluster (by **CoreOS**, retrieved from CoreOS.com)

Keeping State with etcd (Cont.)

In our test setup on the master node, we also run a single node *etcd* key-value store. We can check its content with the `etcdctl` command and see what **Kubernetes** is storing in it:

```
$ systemctl -a | grep etcd etcd2.service loaded active running
etcd2
$ core@master ~ $ etcdctl ls /registry
/registry/ranges
/registry/namespaces
/registry/serviceaccounts
/registry/controllers
/registry/secrets
/registry/pods
/registry/deployments
/registry/services
/registry/events
/registry/minions
/registry/replicasets
```

This gives you a first sneak peek at some of the **Kubernetes** resources: namespaces, pods, deployments, services ...More later - it's a promise!

Networking Setup

Getting all the previous components running is a common task for system administrators that are used to configuration management. But to get a fully functional **Kubernetes** cluster, the network will need to be setup properly as well.

If you have deployed virtual machines (VMs) based on IaaS solutions, this will sound extremely familiar. Containers running on all the nodes will attach to a **Linux** bridge. This bridge will be configured to give IP addresses in a specific subnet and that subnet will be routed to all the other nodes. In essence, you need to treat a container just like a VM. All the containers started on any nodes will need to be able to reach each other.

You can see the detailed explanation about this model here.

The only caveat is that in **Kubernetes**, the lowest compute unit is not a container, but what we call a **POD**. A pod is a group of co-located containers that share the same IP address.

Kubernetes expects this network configuration to be available - it will not do it for you.

This can be achieved at the physical network infrastructure, if you have access to it. Or, it can be achieved with a software defined overlay with solutions like Weave, Flannel, or Calico, among many.

Tim Hockin, one of the lead **Kubernetes** developers, has created a very useful slide deck to understand **Kubernetes** networking. You can check it out here.

Similarities with Mesos

At a high level, there is nothing different between **Kubernetes** and other clustering system.

A central manager exposes an API, a scheduler places the workloads on a set of nodes, and the state of the cluster is stored in a persistent layer.

For example, you could compare **Kubernetes** with Mesos, and you would see the similarities. In **Kubernetes**, however, the persistence layer is implemented with *etcd*, instead of *zookeeper* for **Mesos**.

You should also consider systems like **OpenStack** and **CloudStack**. Think about what runs on their head node, and what runs on their worker nodes. How do they keep state? How do they handle networking? If you are familiar with those systems, **Kubernetes** will not seem that different.

What really sets Kubernetes apart is its features oriented towards fault-tolerance, self-discovery, and scaling, coupled with a mindset that is purely API-driven.

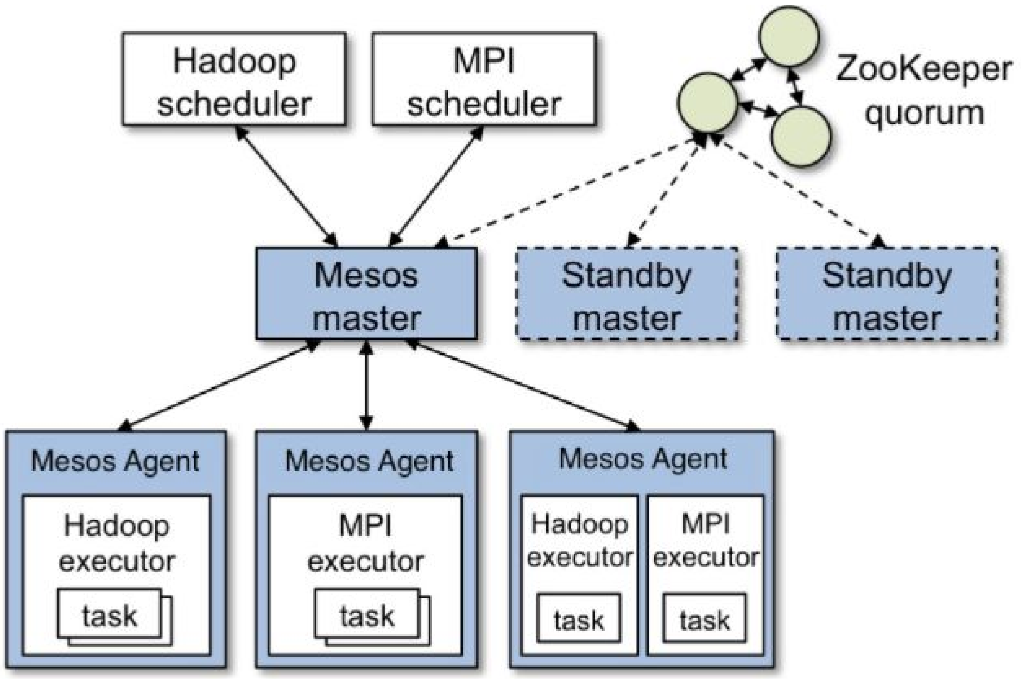


Figure 3.3: Mesos Architecture (by **The Apache Software Foundation**, retrieved from mesos.apache.org)

Kubernetes Architecture Demo

Note: This course includes a screencast demonstrating a Kubernetes head node and a Kubernetes worker. You can access the full demonstration in a private YouTube video.

Choose the correct answer (True or False).

The Kubernetes head node runs an API server.

True

False

Choose the correct answer (True or False).

Kubernetes nodes run the scheduler.

True

False

Choose the correct answer (True or False).

Pods are single container compute units.

True

False

LAB 3

Objective: Review a useful Docker networking feature and start the basic components of the Kubernetes head node.

This lab is made of two activities:

- Use Docker networking to start containers that share the same network stack (i.e network namespace).
- Start a Kubernetes head node with **etcd**, the API server, and the controller manager.

In both activities, you need to be on a Docker host.

Docker networking

Docker allows you to specify different types of networking for your containers. One type that is very handy and used in Kubernetes is the *container* type. It allows you to share a network between two containers. In this exercise, you will illustrate this with three `bash` commands:

On a Docker host, start a `busybox` container that just waits. Then, start another container, using the `--net=container:<other_container_name>` option.

```
$ docker run -d --name=source busybox sleep 3600
$ docker run -d --name=same-ip --net=container:source busybox sleep 3600
```

Now, check the IP address of each container, something like the following should do:

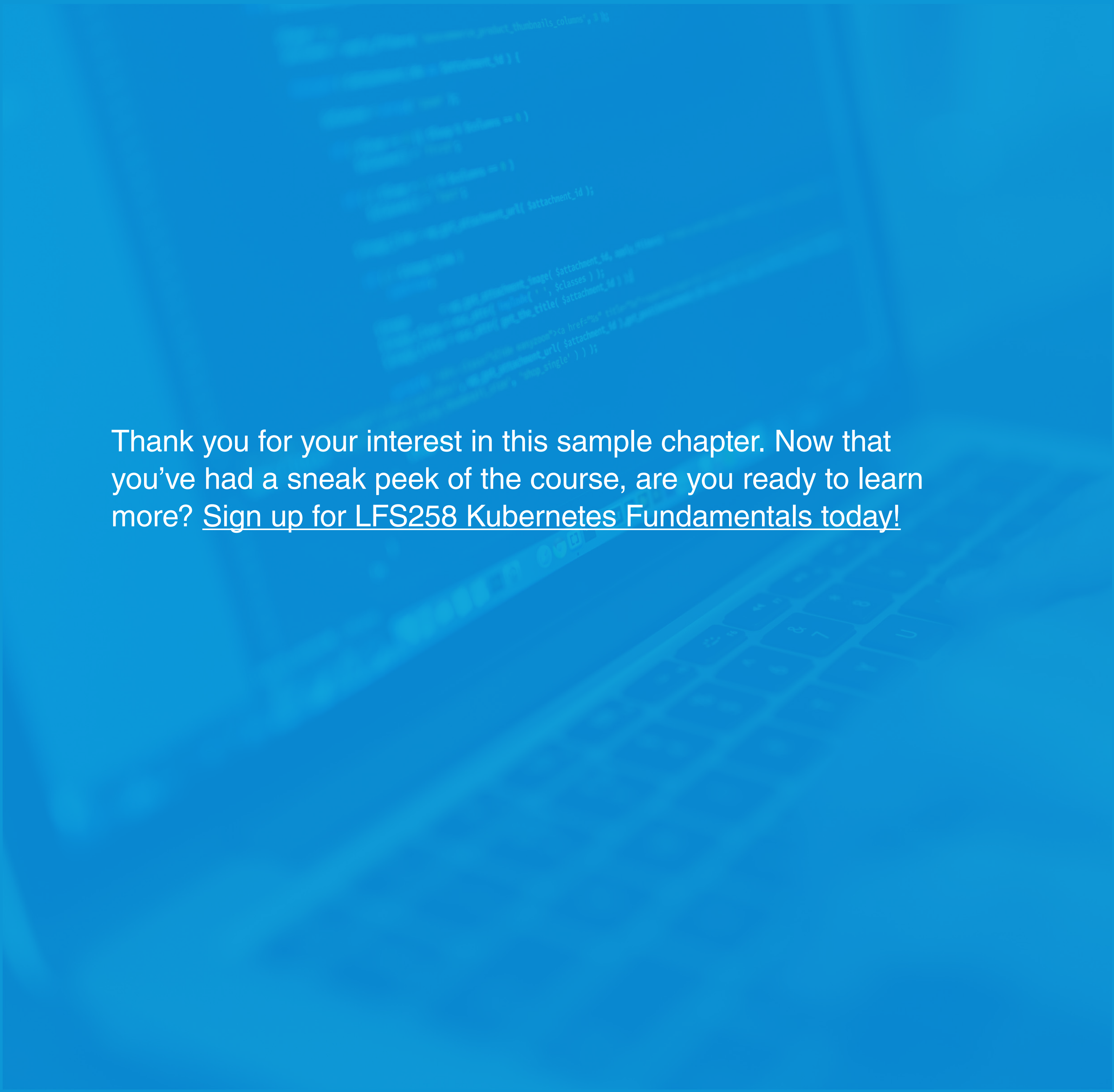
```
$ docker exec -ti same-ip ifconfig
```

You should see that the second container inherited the same IP address as the *source* container. That is because, using the Docker `--net` option, we were able to use the same networking namespace to both containers.

Learning Objectives (Review)

You should now be able to:

- Discuss all the components that make up a Kubernetes cluster.
- Explain the similarity of a Kubernetes cluster to other cluster managers/orchestrators.



THE LINUX FOUNDATION

The Linux Foundation promotes, protects and standardizes Linux by providing unified resources and services needed for open source to successfully compete with closed platforms.

To learn more about our Linux Training program, please visit us at training.linuxfoundation.org.